11.0.—Reconocimiento de librerías utilizando firmas FLIRT

Vamos a explorar que ocurre en IDA una vez se nos muestra la siguiente indicación **"The initial autoanalysis has been finished"** (IDA nos genera esta indicación en la ventana de mensajes cuando ha finalizado el proceso automatizado de análisis de un binario nuevo una vez ha sido cargado).

Cuando realizamos ingeniería inversa en cualquier binario, lo que se intenta es perder el menos tiempo posible en comprender el comportamiento y funcionamiento de las librerías leyendo el código fuente, si lo más fácil para saber como funcionan éstas es, remitiéndonos a su página de explicación, estudiando algún código fuente o investigando en Internet. El desafío en un binario enlazado estáticamente es poder vincular y distinguir entre dos códigos, el código de la aplicación y el código de la librería. En un binario de este tipo, librerías enteras están combinadas con el código de la aplicación formando un único archivo ejecutable. Afortunadamente para nosotros, IDA dispone de herramientas que le habilitan reconocer y marcar el código de librerías, permitiéndonos enfocar nuestra atención solamente a un único código, la aplicación.

11.1.—Tecnología de identificación y reconocimiento rápido de librería.

Fast **L**ibrary **I**dentification and **R**ecognition **T**echnology

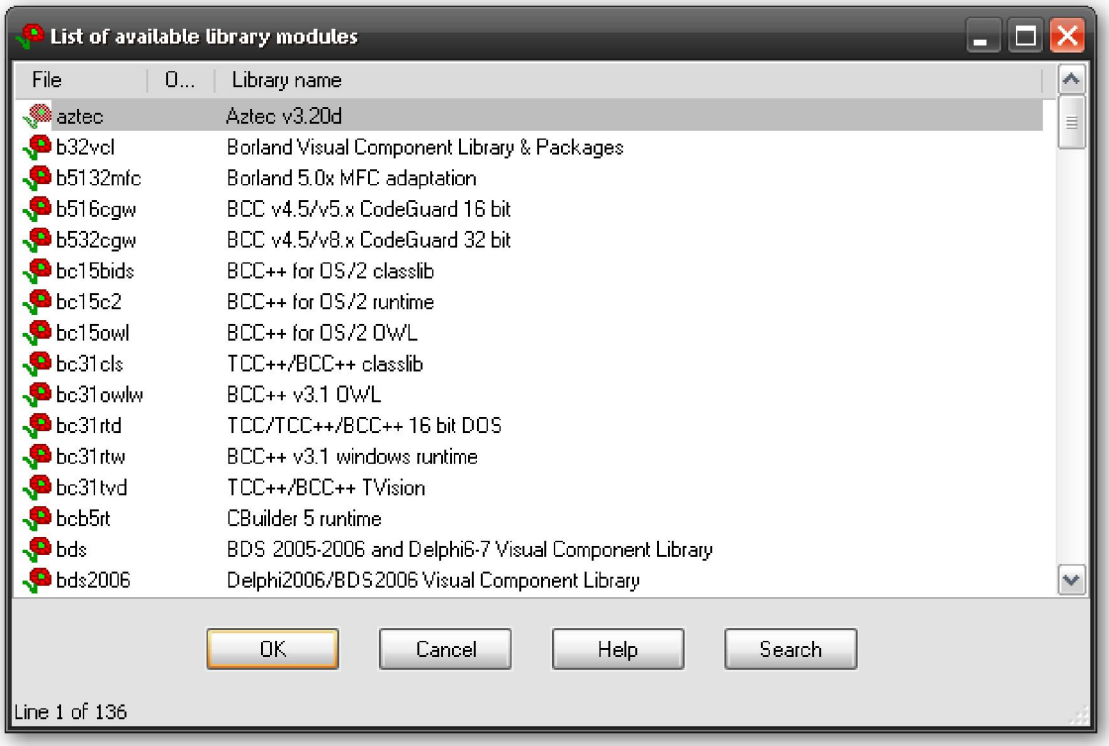
La tecnología de identificación y reconocimiento rápido de librería, mas conocido como **FLIRT**, abarca el conjunto de técnicas empleadas por IDA para identificar sucesiones de código como código de librería. El corazón de FLIRT son los algoritmos de coincidencia con ciertas pautas, los cuales habilitan a IDA el poder determinar rápidamente si una función desensamblada coincide con una de las muchas firmas conocidas por IDA. El directorio **Archivos de programa\IDA\sig**, contiene los archivos de firmas que se adjuntan con la versión de IDA. La mayor parte, son librerías compiladas con compiladores comunes de Windows, sin embargo también existen otras que no son firmas Windows.

Los archivos de firma utilizan un formato de encabezado IDA a medida, en el cual son adjuntados y comprimidos los datos de la firma. En la mayoría de los casos no nos indican el tipo de librería asociada a cada archivo de firma. Depende de cómo hayan sido creados estos, pueden contener un comentario donde se explica el nombre de librería y su contenido. El visionado de algunas de las primeras líneas extraídas en forma ASCII del archivo de firma, pueden revelarnos su contenido. Las siguientes órdenes Unix, **strings** (escrito 1) y **head**, la cual se utiliza para visionar solamente las primeras líneas de código fuente, se utilizan para poderlas ver. Generalmente el comentario revelador está situado en la segunda o tercera línea:

 # strings archivo_sig | head -n 3

Dentro de IDA tenemos dos opciones para poder ver los comentarios asociados con los archivos de firma. Primero, podemos acceder a la lista de firmas que han sido aplicadas a un binario realizando la acción **View > Open Subviews > Signatures**.

Segundo, la lista de todos los archivos de firmas pueden mostrarse como parte de un proceso manual de elección de firmas para la aplicación, realizando la acción **File > Load File > FLIRT signature file.**



11.2.—Aplicación de firmas FLIRT

Cuando un binario es abierto por primera vez, IDA intentará aplicar archivos de firmas especiales para dicho binario, designará como se iniciarán las firmas en el punto de entrada **"entry point"** del binario. Como que el código del punto de entrada generado por distintos compiladores difieren entre ellos, para saber a que firma corresponde se utiliza una técnica para identificar el compilador que se ha utilizado para genrar dicho binario.

Kit-kat teórico: función MAIN versus función _START

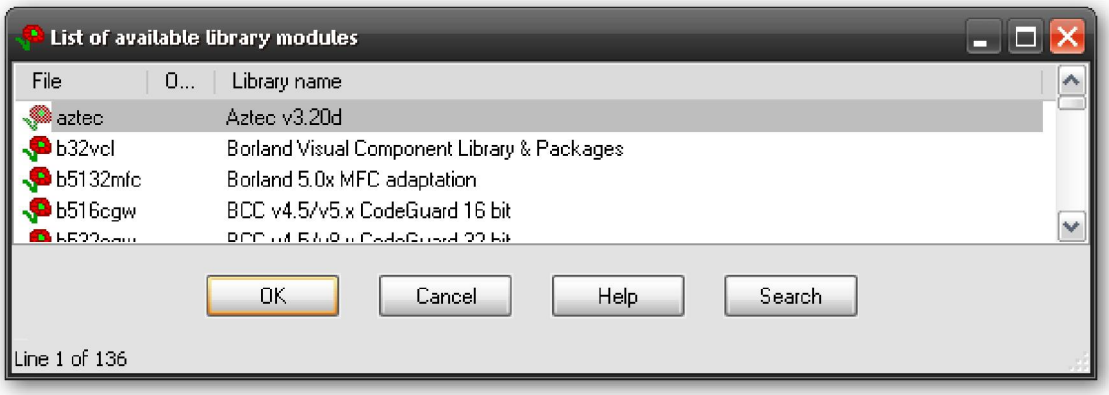
Recordemos que el punto de entrada de un programa es la dirección de la primera instrucción que será ejecutada. Los programadores en C, incorrectamente creen que el punto de entrada del programa es la dirección de la función llamada **main**, cuando de hecho no es así. El tipo de archivo del programa, no el lenguaje utilizado para crear el programa, es el que dicta a través de la línea de órdenes la forma en que son suministrados los argumentos a un programa. Para corregir cualquier diferencia entre la forma en que el cargador presenta los argumentos y la forma en que el programa espera recibirlos, por ejemplo a través de los parámetros a **main**, debe ejecutarse un código de inicialización antes de transferir el control a **main**. Es dicha inicialización la que IDA designa como **entry point** del programa y lo etiqueta como **_start**.

Este código de inicialización, también es el responsable de cualquier tarea de inicialización antes de que se permita la ejecución de **main**. En un programa C++, este código es el responsable de asegurarse de que se llamen a los constructores globales para declarar los objetos antes de la ejecución de **main.** De forma similar, se inserta un código de limpieza después de que **main** se haya ejecutado para invocar a los destructores de los objetos globales antes de que se termine la ejecución del programa.

Si IDA identifica el compilador utilizado para crear un binario, entonces el archivo de firma correspondiente a las librerías del compilador se carga y se aplica al resto del binario. Las firmas adjuntadas con IDA tienden a dar prioridad a compiladores como **Microsoft Visual C++** o a **Borland Delphi**. La razón de esta actuación es que dichos compiladores adjunta un número finito de librerías. Para los compiladores de código abierto, como **GNU gcc**, las variaciones de las librerías asociadas a un binario son tan numerosas como sistemas operativos de los compiladores. Estas variaciones producen distinto código compilado, con lo cual diferentes opciones de compilación nos proporcionan distintos códigos compilados, y aunque IDA nos proporciona distintas firmas de librerías de compiladores de código abierto no es práctico.

Bajo estas circunstancias puede ser que necesitemos aplicar manualmente firmas a una base de datos. En ocasiones puede ocurrir que IDA identifique el compilador utilizado para crear el binario pero no tenga las librerías correspondientes al compilador. En estos casos, necesitarás trabajar sin firmas o necesitarás obtener copias de las librerías estáticas utilizadas por el binario y generar nuestras propias firmas. Otras veces, IDA sencillamente puede fallar en la identificación del compilador, haciendo imposible determinar qué firmas se tienen que aplicar a la base de datos. Esto suele suceder cuando analizamos código ofuscado en el cual las rutinas de inicialización han sido manipuladas para no poder identificar el compilador. Lo primero que hay que hacer en estos casos, es "des-ofuscar" suficientemente el binario con la esperanza de que coincida con alguna librería de firma. La técnica para trabajar con código ofuscado la trataremos más adelante.

Si queremos aplicar firmas manualmente a una base de datos, como hemos dicho antes, lo haremos con la acción **File > Load File > FLIRT Signature File**, lo cual nos proporcionará un diálogo de selección de firmas mostrado en la figura.



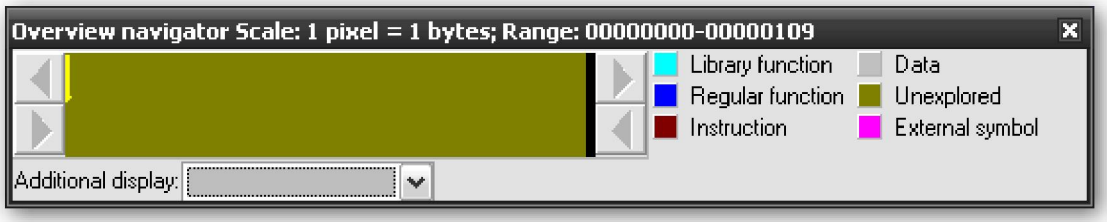
La columna **File** refleja el nombre de cada archivo **.sig** del directorio **Archivos de programa\IDA\sig**. Observemos que no existe otra alternativa para indicar otros directorios de archivos **.sig**. Si en alguna ocasión generamos alguna firma, la tendremos que colocar dentro de dicho directorio con las otras firmas. La columna **Library name** muestra el comentario adjuntado dentro de cada archivo con el nombre de la librería. Tengamos presente que dichos comentarios sólo describen lo que el creador de la firma pretenda indicar.

Cuando un módulo de librería es seleccionado, las firmas contenidas en el correspondiente archivo **.sig** son cargadas y comparadas para cada función dentro de la base de datos. Sólo puede aplicarse un conjunto de firmas cada vez, con lo cual repetirá dicho proceso por cada conjunto de firmas que se apliquen a la base de datos. Cuando se encuentra una función que coincide con una firma, la función es marcada como una función de librería, y dicha función es renombrada automáticamente de acuerdo a la firma con la que haya coincidido.

Atención: Solamente las funciones nombradas por IDA como **Dummy name**, pueden ser renombradas automáticamente. En otras palabras, si hemos renombrado una función y

posteriormente dicha función coincide con una firma, entonces dicha función no será renombrada como resultado de su coincidencia con una firma. Por lo tanto es necesario aplicar las firmas cuanto antes, en el proceso de análisis.

Recordemos otra vez que el problema de un binario enlazado estáticamente es el poder distinguir entre el código de la aplicación y el código de librería. Si tenemos la suerte de tener un binario enlazado estáticamente del cual no hayan despojado sus símbolos, podremos utilizar los nombres de función para ayudarnos a identificarlas en el código. Sin embargo si el binario ha sido despojado de sus símbolos, quizás tengamos centenares de funciones, con nombres generados por IDA pero sin indicarnos lo que realiza cada función. En ambos casos, IDA será capaz de identificar las funciones de librería sólo si las firmas están disponibles. En la siguiente figura mostramos la **Overview Navigator** de un binario enlazado estáticamente.



Vemos que no existen funciones que se puedan identificar como funciones de librería. Después de aplicarse las firmas adecuadas el **Overview Navigator** se transforma en, figura abajo.

Ahora podemos ver, que el **Overview Navigator** nos proporciona mejor indicado que efectivamente hay un conjunto de firmas. Con una gran proporción de coincidencia de firmas.

Tenemos dos puntos a recordar cuando apliquemos firmas. En primer lugar, las firmas son útiles cuando trabajan con un binario que no haya sido despojado de sus símbolos, en cuyo caso las firmas se utilizan más para ayudar a IDA a identificar las funciones de librería que para renombrar a dichas funciones. En segundo lugar, los binarios enlazados estáticamente pueden estar vinculados a varias librerías separadas, por lo cual dicha aplicación requerirá varios conjuntos de firmas para identificar completamente las funciones de librería. Con cada aplicación de distintas firmas, aparecerán sectores nuevos en el **Overview Navigator** y este se irá transformando para reflejar el nuevo código de librería encontrado.

11.3.—Crear archivos de firma FLIRT

Como ya hemos dicho antes, es prácticamente imposible que IDA adjunte todos los archivos de firmas para todas las librerías estáticas existentes. Por eso mismo, IDA proporciona a sus usuarios herramientas e información necesaria para crear nuestras propias firmas, Hex Rays nos proporciona un conjunto de herramientas, **Fast Library Acquisition for Identification and Recognition (FLAIR)**, para poderlo realizar. Las herramientas FLAIR se encuentran en el CD de IDA o podemos descargarlas del web de Hex-Rays, solo para usuarios autorizados. Como otros añadidos de IDA se distribuye un formato .zip. Para la versión IDA 5.2, la herramienta asociada el **flair52.zip**.

Su instalación es simple, extraemos el contenido del archivo zip, se recomienda descomprimirlo en un directorio con nombre **flair**. Y una vez descomprimido existen varios archivos de texto que constituyen la documentación de dicha herramienta. Los archivos más interesantes son:

readme.txt
Se explica el proceso de la creación de firmas.

plb.txt
Este archivo describe la utilización de la librería estática de análisis, **plb.exe**. La librería analizadora la explicaremos en la próxima sección con más detalle.

pat.txt
Este archivo detalla el formato de los archivos maestros, los cuales representan el primer paso del proceso de creación de firma. Los archivos maestros se describirán en la siguiente sección.

sigmake.txt
Este archivo detalla el uso de sigmake.exe para generar archivos **.sig** a partir de los archivos maestros. Con más detalle lo veremos en la siguiente sección.

Otro contenido de interés se encuentra en el directorio **bin**, el cual contiene todos los archivos ejecutables de las herramientas FLAIR, y el directorio **startup** el cual contiene los archivos maestros de las secuencias de inicio más comunes asociadas a distintos compiladores que nos dan como salida tipos de archivo como: **PE, ELF** y otros. Un punto importante a tener en cuenta de las herramientas **FLAIR**, es que aunque éstas sólo se ejecuten desde la consola de Windows, los archivos de firma resultantes pueden utilizarse en todas las variantes de IDA (Windows, Linux y OS X).

11.3.1.—Pasos generales para crear firmas

El proceso básico para crear firmas no es complicado y sólo hay que seguir estos pasos:

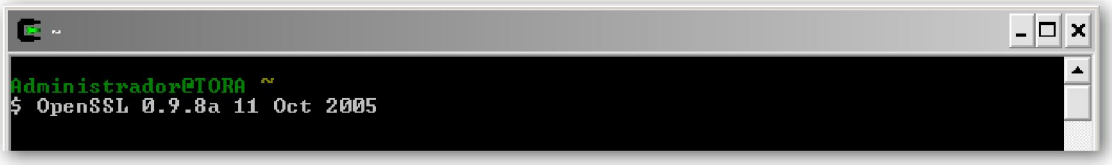
1. Obtenemos una copia de la librería estática para la cual queremos crear el archivo de firma.
2. Utilizamos uno de los analizadores FLAIR para crear el archivo maestro de la librería.
3. Ejecutamos **sigmake.exe**, éste nos procesa el archivo maestro y nos genera un archivo de firma.
4. Instalamos el nuevo archivo **.sig** en IDA copiándolo en la carpeta **sig**.

Por desgracia, sólo el último paso es tan fácil como parece. En los siguientes párrafos, estudiaremos los tres primeros pasos con más detalle.

11.3.2.—Identificar y encontrar las librerías estáticas

El primer paso en el proceso de generación de firmas es localizar una copia de la librería estática de la cual queremos generar las firmas. Esto puede ser un poco difícil por varias razones. El primer obstáculo es determinar qué biblioteca se necesita en realidad. Si el binario que vamos a analizar no ha sido despojado de sus símbolos, podemos tener suerte de tener los nombres reales de sus funciones en el desensamblado, en cuyo caso “San Google” seguramente nos proporcionará varios candidatos.

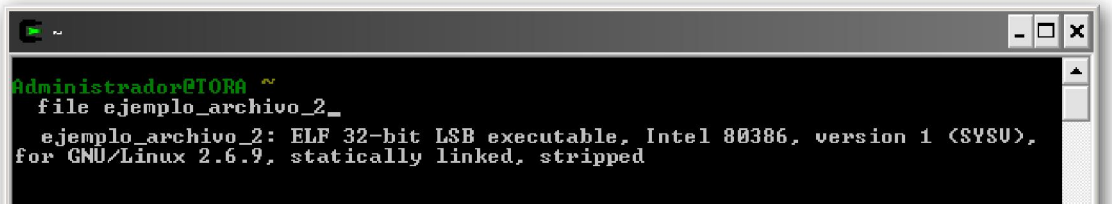
Si el binario ha sido despojado de sus símbolos, entonces ya no es tan fácil. Pero como carecemos de los nombres de funciones, podemos encontrar **strings** que nos pueden dar la pista para identificar la librería, el siguiente ejemplo es evidente:



Los Copyright y las cadenas de error normalmente son suficientes para determinarla y seguidamente utilizaremos a “San Google” para corroborarlo. Si lo que elegimos es ejecutar la órden **strings** desde la línea de órdenes, acordémonos de utiliza la opción **–a**, la cual fuerza la búsqueda de **strings** para todo el binario; sino es así podemos perder cadenas de datos.
En el caso de librerías de código abierto, probablemente encontremos el código fuente sin ningún tipo de problema. Por desgracia, aunque el código fuente nos sea útil para comprender el funcionamiento de los binarios, no podremos utilizarlo para generar sus firmas. Lo que sí podemos hacer es utilizar el código fuente para construir nuestra propia versión de librería estática y entonces utilizar dicha versión en el proceso de generación de firma. Sin embargo, probablemente existirán variaciones en la estructura resultante de nuestra librería y la librería que estemos analizando, ya que las firmas generadas no serán exactamente iguales.

La mejor opción para determinar el origen exacto de un binario en cuestión, es conocer exactamente el sistema operativo, su versión y su distribución (si la hay). Con dicha información, la mejor opción para crear firmas es copiar las librerías en cuestión desde un sistema configurado idénticamente. Naturalmente, esto nos conduce a un nuevo desafío: ¿En qué sistema fue creado dicho binario? El primer paso es utilizar la utilidad **file** para obtener alguna información preliminar respecto al binario en cuestión. En muchos casos esta información será suficiente para proporcionarnos los sistemas operativos candidatos. El siguiente ejemplo es una salida específica de **file**:

En este caso podemos afirmar directamente que es un sistema FreeBSD 5.4 por lo tanto buscaremos **libc.a**. En contraposición el siguiente ejemplo es más ambiguo:



Podemos centrarnos en un sistema Linux, pero debido a las muchas distribuciones existentes no nos dice mucho. Si efectuamos la orden **strings** hallaremos lo siguiente:

GCC: (GNU) 4.1.1 20060525 (Red Hat 4.1.1-1)

Con nuestra búsqueda hemos hallado que es una distribución Red Hat (o derivadas) viene adjuntada con la versión 4.1.1 de gcc. La etiqueta GCC no es común en los

binarios compilados utilizando **gcc**, pero afortunadamente esta ha sobrevivido al proceso de despojo de símbolos y permanece visible para **strings**.

11.3.3.—Crear archivos maestros

En este momento, deberíamos tener una o más librerías de las que queremos crear firmas. El siguiente paso es crear un archivo maestro para cada librería. Los archivos maestros se crean utilizando la utilidad analizadora de FLAIR, apropiada. Al igual que los archivos ejecutables, los archivos de librerías están construidos con distintas especificaciones de formato de archivo. FLAIR nos proporciona distintos analizadores de los más comunes formatos de archivos de librería. Como se indica en el **readme.txt** de FLAIR, los siguientes analizadores pueden encontrarse en el directorio **FLAIR\bin**:

plb.exe
Analizador para librerías **OMF** (normalmente utilizado por compiladores Borland)

pcf.exe
Analizador para librerías **COFF** (normalmente utilizado por compiladores Microsoft)

pelf.exe
Analizador para librerías **ELF** (se encuentra en cualquier sistema Unix)

ppsx.exe
Analizador para librerías Sony PlayStation **PSX**

ptmobj.exe
Analizador para librerías **TriMedia**

pomf166.exe
Analizador para archivos objeto de **Kiel OMF**

Para crear un archivo maestro de cierta librería, especificaremos el analizador correspondiente al formato de la librería, el nombre de la librería que queremos analizar y el nombre del archivo maestro el cual debemos haber generado. Como ejemplo para una copia de **libc.a** desde un sistema **FreeBSD 6.1**, deberíamos utilizar:

Como podemos ver, el analizador nos indica que el archivo que se ha analizado es **libc.a**, el número de funciones que se han saltado **(0)** y el número de secuencias de firmas que se han generado **(986)**. Cada analizador acepta un conjunto de opciones de órdenes ligeramente distintas, las cuales están documentadas en la utilización de cada analizador. Si ejecutamos cualquier analizador sin ningún argumento, se nos mostrará la lista de opciones de órdenes aceptadas por el analizador. El archivo **plb.txt** contiene información detallada sobre las opciones aceptadas por el analizador **plb.exe**. Estos archivos son una muy buena fuente de información respecto a los analizadores. En la

mayoría de los casos sólo con el nombre de la librería a analizar y el archivo maestro son suficientes para generar las firmas.

Un archivo maestro, es un archivo **txt** que contiene, una por línea, las secuencias extraídas de una librería analizada las cuales representan sus funciones. Veamos algunas líneas de un archivo maestro creado previamente:

```
5589E58B55108B450C8B4D0885D2EB06890183C1044A75F88B4508C9C3..... 00 0000 001D :0000
_wmemset
5589E58B4D1057C1E102568B7D088B750CFCC1E902F3A55E8B45085FC9C3.... 00 0000 001E :0000
_wmemcpy
5589E556538B751031DB39F38B4D088B550C73118B023901751183C10483C204 19 A9BE 0039 :0000
_wmemcmp
```

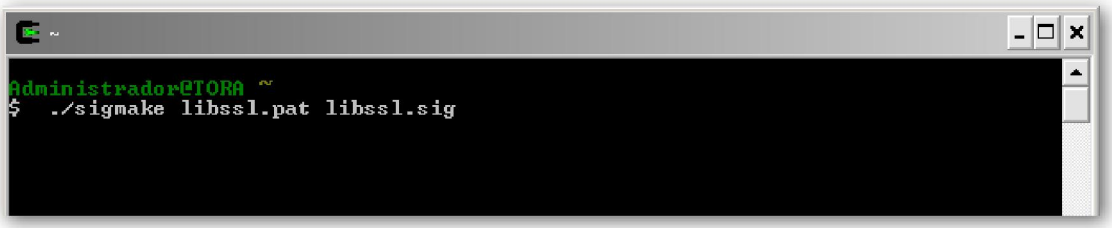
En el archivo de FLAIR **pat.txt**, se describe un modelo de formato. Resumiendo, la primera parte es el listado de la secuencia inicial de bytes de la función, hasta un máximo de 32 bytes. La cantidad de bytes mostrados variará según las entradas, los espacios vacíos se rellenaran con puntos como podemos ver en **_wmemset**. Después de los primeros 32 bytes, se proporciona información adicional para dar más precisión al proceso de coincidencia de firma. La información adicional codificada en cada línea de secuencia incluye un valor **CRC16** (Esto es un valor de 16-bit de comprobación de redundancia cíclica. La ejecución del CRC16 utilizado para la generación de la secuencia se encuentra en **crc16.cpp** de la utilidad FLAIR), calculado con una parte de la función, el largo de la función en bytes y una lista de los nombres de símbolos referenciados por la función. Por lo general, las funciones más largas que referencia a muchos símbolos producen las líneas de secuencia más complejas.

Distintos programadores, han creado utilidades diseñadas para generar secuencias desde las bases de datos de IDA. Una de estas utilidades es **IDB_2_PAT**, es un plugin programado por **J.C.Roberts** el cual es capaz de generar las secuencias de una o más funciones que existan en una base de datos. Utilidades como esta se utilizan para encontrar código similar en bases de datos de las cuales no tenemos acceso al archivo original de la librería utilizada para crear el binario analizado.

11.3.4.—Crear archivos de firmas

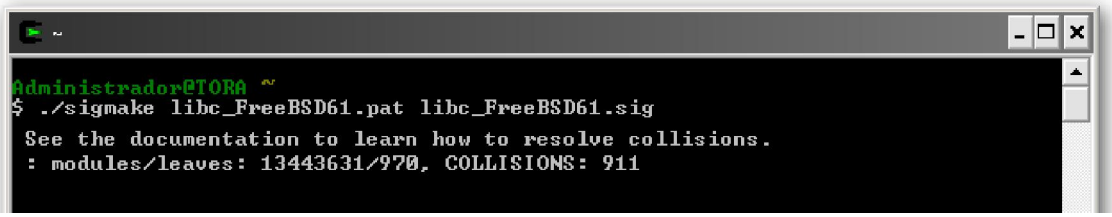
Una vez que hemos creado el archivo maestro de una librería en particular, el siguiente paso del proceso para crear las firmas es generar el archivo **.sig** el cual se podrá utilizar en IDA. El formato de un archivo de firma IDA es substancialmente distinto a un archivo maestro. Los archivos de firmas utilizan un formato binario propio diseñado para minimizar la cantidad de espacio requerido para representar toda la información presente en una línea de secuencia de bytes para permitir la máxima eficiencia en encontrar la coincidencia de las firmas con el contenido de la base de datos escogida. La descripción de la estructura de un archivo de firma lo podemos encontrar en el website de Hex-Rays.

La utilidad de FLAIR, **sigmake.exe** es utilizada para crear los archivos firma a partir de los archivos maestros. Debemos tener en cuenta que la generación del archivo maestro y la generación del archivo de firma, son dos fases completamente independientes, lo cual nos permite utilizar otras utilidades para generar los archivos maestros. Por lo tanto la generación de la firma sólo se realizará con **sigmake.exe** el cual analizará un archivo **.pat** para crear un archivo **.sig**, como podemos ver:



Si todo va bien, se crea un archivo **.sig**, listo para instalarlo en la carpeta **sig** de IDA. Sin embargo, el proceso raramente funciona sin ningún problema.

La generación de firmas es a menudo un proceso iterativo, por eso durante esta fase se tienen que controlar las **"collisions"(**coincidencias). Una **"collision"** ocurre cada vez que dos funciones tienen una secuencia de bytes idéntica. Si la coincidencia no se puede resolver de alguna forma, no es posible determinar qué función es actualmente confrontada durante el proceso de firmas de la aplicación. Por lo tanto, **sigmake** debe ser capaz de resolver cada firma generada exactamente con el nombre de una función. Cuando esto no es posible, por la presencia de secuencias idénticas de una o más funciones, **sigmake** rechaza la creación del archivo **.sig** y genera un archivo de exclusiones **.exc**. El primer paso utilizando **sigmake** podemos verlo seguidamente:



La documentación a la que se refiere es **sigmate.txt**, la cual describe su uso y el proceso para solucionar las coincidencias. En realidad, cada vez que se ejecuta, este busca los archivos de exclusiones correspondientes para tomar información de cómo resolver todas las coincidencias que **sigmake** pueda encontrar en el proceso de nombramiento de las funciones del archivo maestro. En la ausencia de tales archivos de exclusión, y cuando ocurren coincidencias, genera dichos archivos de exclusiones antes que el de las firmas. En el ejemplo anterior, nos encontraremos con un archivo nuevo llamado **libc_FreeBSD61.exc**. Antes de ser creado, los archivos de exclusiones, archivos de texto los cuales detallan los conflictos que sigmake encontró durante el proceso del archivo maestro, deben de ser editados para proporcionar a **sigmake** una guía de cómo deberá resolver cualquier conflicto en las secuencias de las funciones. El proceso para editar un archivo de exclusiones es el siguiente.

Cuando ha sido generado por **sigmake**, todo archivo de exclusiones empieza con las líneas siguientes:

```
;--------- (delete these lines to allow sigmake to read this file)
; add '+' at the start of a line to select a module
; add '-' if you are not sure about the selection
; do nothing if you want to exclude all modules
```

Estas líneas intentan recordarnos que hay que resolver las coincidencias antes de realizar la generación de firmas. Lo principal es borrar estas cuatro líneas con o sigmake no realizará el análisis del archivo de exclusiones durante su ejecución. El próximo paso

es informar a sigmake que queremos solucionar las coincidencias. Algunas líneas extraidas de **libc_FreeBSD61.exc** se muestran a continuación:

```
___ntohs            00 0000 0FB744240486C4C3........................................
___htons            00 0000 0FB744240486C4C3........................................

_index              00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3.........
_strchr             00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3.........

_rindex             00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3......
_strrchr            00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3......
```

Estas líneas detallan tres coincidencias separadas. En este caso nos está diciendo que la función **ntohs** no se puede distinguir de **htons**, **index** tiene la misma firma que **strchr** y **rindex** coincide con **strrchr**. Si observamos dichas funciones, el resultado no debe sorprendernos, ya que las funciones coincidentes son esencialmente idénticas, por ejemplo **index** y **strchr** ejecutan la misma acción.

Con el fin de que nosotros decidamos, sigmake espera que designemos una sola función de cada grupo asociada a la firma. Seleccionaremos una función anteponiendo al nombre el carácter (+) si queremos que dicho nombre se aplique a cualquier función que coincida con la firma o un carácter (-) si simplemente queremos que se añada como un comentario a la base de datos cada vez que su correspondiente firma coincida con otra. Si no queremos que se le aplique ningún nombre en la base de datos a la correspondiente firma coincidente, entonces no añadimos ningún carácter. El siguiente listado sería una solución válida de las coincidencias mostradas antes:

```
+___ntohs            00 0000 0FB744240486C4C3........................................
___htons            00 0000 0FB744240486C4C3........................................

_index              00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3.........
_strchr             00 0000 538B4424088A4C240C908A1838D974074084DB75F531C05BC3.........

_rindex             00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3......
-_strrchr           00 0000 538B5424088A4C240C31C0908A1A38D9750289D04284DB75F35BC3......
```

En este caso hemos elegido utilizar el nombre **ntohs** siempre que la primera firma coincida, no hacer nada cuando se coincida con la segunda firma y añadir un comentario respecto a **strrchr** cuando coincida con la tercera firma. Los siguientes puntos son útiles para intentar resolver coincidencias:

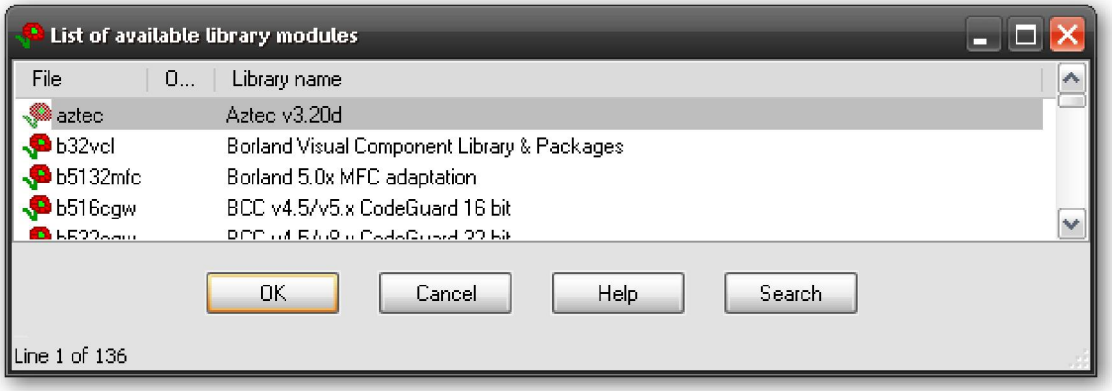
1. Para ejecutar la mínima resolución de coincidencias, simplemente borra las cuatro líneas de comentarios al inicio de los archivos de exclusiones.
2. Nunca añadas un carácter + o – a más de una de las funciones coincidentes del grupo.
3. Si un grupo de coincidencias contiene sólo una función, no añadas + o -, simplemente déjala.
4. Los subsecuentes fallos de sigmake producen datos, incluyendo líneas de comentarios que serán añadidos a los archivos de exclusiones. Estos datos tendrán que eliminarse y corregir los datos originales, si los datos ya son corregidos, sigmake no tiene porque fallar cuando volvamos a ejecutarlo.

Una vez hayamos realizados los cambios apropiados en nuestro archivo de exclusiones, deberemos guardar dicho archivo y reejecutar sigmake utilizando los mismos argumentos iniciales. La segunda vez sigmake tendrá que guiarse con nuestro archivo de exclusiones creando de esta forma un archivo **.sig** sin problemas. Si ha tenido éxito

lo notaremos por la falta de mensajes de error y la presencia de nuestro archivo **.sig**, como se muestra seguidamente:

Una vez que el archivo de firma ha sido generado, se lo proporcionaremos a IDA copiándolo en el directorio **sig.** Con lo cual la nueva firma estará a nuestra disposición realizando la acción **File > Load File > FLIRT Signature File**.

Tengamos en cuenta también, todas las opciones que podemos adjuntar tanto al generar el archivo maestro como a sigmake. Una explicación detallada de las opciones disponibles lo encontraremos en **plb.txt** y **sigmake.txt**. La única opción que resaltaremos es la opción **–n** utilizada en sigmake. Esta opción permite adjuntar un nombre descriptivo dentro del archivo de firma generado. Este nombre se mostrará en el proceso de selección de firma, figura siguiente, el cual puede ser muy útil cuando clasifiquemos las la lista de firmas.



La siguiente línea de órdenes adjuntará la cadena del nombre, "**FreeBSD6.1 Lib C estandar**" dentro del archivo de firma generado:

11.3.5.—Firmas de arranque

IDA también reconoce una forma especial de firmas, son las llamadas firmas de arranque. Dichas firmas se aplican cuando un binario es guardado por primera vez en una base de datos intentando identificar el compilador que se utilizó para crear el binario. Si IDA identifica al compilador, los archivos de firma asociados a dicho compilador son automáticamente cargados durante el análisis inicial del binario.

Cuando por primera vez se carga un binario y el compilador no se reconoce, las firmas de arranque se seleccionan de acuerdo al tipo de archivo del binario cargado. Por ejemplo, si se está cargando un binario Windows PE, se cargarán las firmas de arranque específicas a los binarios PE, como un esfuerzo para determinar el compilador utilizado para crear dicho binario PE.

Con el fin de crear firmas de arranque, **sigmake** procesa secuencias descritas en una rutina de arranque,… (La rutina de arranque, generalmente está diseñada como punto de entrada del programa. En un programa **C/C++**, el propósito de la rutina de arranque es inicializar el entorno del programa antes de pasar el control a la función **main)**…, generada por varios compiladores y grupos de firmas específicas al tipo de archivo. En el directorio **startup** de FLAIR, se encuentran las secuencias utilizadas por IDA juntamente con el script **startup.bat,** utilizado para crear las firmas de arranque correspondientes a nuestras secuencias. En los ejemplos de cómo utilizar sigmake nos podemos referir a **startup.bat** para aprender como crear firmas de arranque para un formato específico.

En el caso de los archivos PE, observaremos varios archivos **pe_*.pat** en el directorio **startup**, los cuales describen las secuencias utilizadas por los compiladores más comunes de Windows, **pe_vc** para secuencias **Visual Studio** y **pe_gcc.pat** para secuencias **Cygwin/gcc**. Si deseamos añadir secuencias adicionales para archivos PE, podemos añadir alguna de las secuencias existentes o crear un archivo de secuencias nuevo, poniendo el prefijo **pe_** al script de generación de firma de arranque para de esa forma encontrar correctamente sus secuencias e incorporar las nuevas firmas PE generadas.

Por último y concerniente al formato de las secuencias de arranque, ligeramente distintas a las generadas para las funciones de librería, su diferencia estriba en el hecho de que la línea de secuencia de arranque es capaz de relacionar la secuencia con otros conjuntos de firmas que podrán también ser aplicadas para comprobar su coincidencia con el modelo. Apuntar también que aparte de los ejemplos de secuencias de arranque incluidos en el directorio **startup**, no existe más documentación sobre ello en FLAIR.

Performance Bigundill@